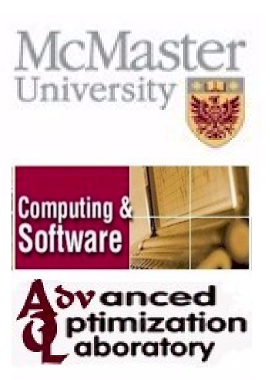
Suffix-based text indices, construction algorithms, and applications.

F. Franek
Computing and Software
McMaster University
Hamilton, Ontario

2nd CanaDAM Conference
Centre de recherches
mathématiques in Montréal

May 25-28, 2009



inverted file (**postings file** or **inverted index**) is an index data structure storing a mapping from words, to its membership in the given text (or a set of texts). The purpose is to allow word search.

T_0 = CanaDAM is a great conference

T_1 = A conference really great

T_2 = Best conference

vocabulary (small space)			*occurences (big space)*
	Canadam	{0}	
	is	{0}	
	a	{0,1}	
	great	{0,1}	
	conference	{0,1,2}	
	really	{1}	
	best	{2}	

Searching for great CanaDAM: $\{0,1\} \cap \{0\} = \{0\}$ directs us to the text T_0

Not all words might be stored, not all forms of words might be stored (lower/upper case, plurals, etc.)

May contain "address"

vocabulary (small space)			*occurences (big space)*
	Canadam	{0 }	
	is	{0 }	
	a	{0 ,1 }	
	great	{0 ,1 }	
	conference	{0 ,1 ,2 }	
	really	{1 }	
	best	{2 }	

To reduce space, blocking may be used (fewer pointers, so pointers can be smaller)

T0 = | CanaDAM is a | great conference |

T1 = | A conference | really great |

T2 = | Best conference |

vocabulary (small space)		*occurences (big space)*
Canadam	{0 }	
is	{0 }	
a	{0 ,1 }	
great	{0 ,1 }	
conference	{0 ,1 ,2 }	
really	{1 }	
best	{2 }	

Index	Small collection (1 Mb)		Medium collection (200 Mb)		Large collection (2 Gb)	
Addressing words	45%	73%	36%	64%	35%	63%
Addressing documents	19%	26%	18%	32%	26%	47%
Addressing 64K blocks	27%	41%	18%	32%	5%	9%
Addressing 256 blocks	18%	25%	1.7%	2.4%	0.5%	0.7%

Table 8.1 Sizes of an inverted file as approximate percentages of the size the whole text collection. Four granularities and three collections are considered. For each collection, the right column considers that stopwords are not indexed while the left column considers that all words are indexed.

stopword = frequently occurring words that carry no meaning e.g. the a an

Building an inverted file is a relatively cheap task: $O(n)$

vocabulary can be maintained as *hash table*, *trie*, or *B-tree*, however a simple list in lexicographic order is better for space and competitive in search (binary search $O(\log n)$)

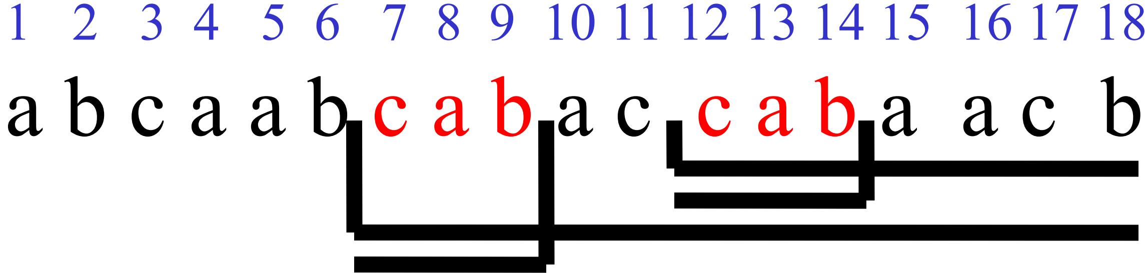
Practical figures show space requirement and text to be traversed sublinear (close to $O(n^{0.85})$). No other index can do that.

Disadvantage - queries for phrases are expensive to perform. Since the text is viewed as a sequence of words, *no subword search* possible (not good, for instance, for analyses of DNA or protein sequences).

Suffix-based indices: suffix trees and suffix arrays allow equal retrieval of any subword -- suitable for ***substring problem***.

The task is to identify all occurrences of a substring fast and efficiently. Instead of re-scanning the string every time we are looking for a pattern, we “prepare” a data structure to do the search easily.

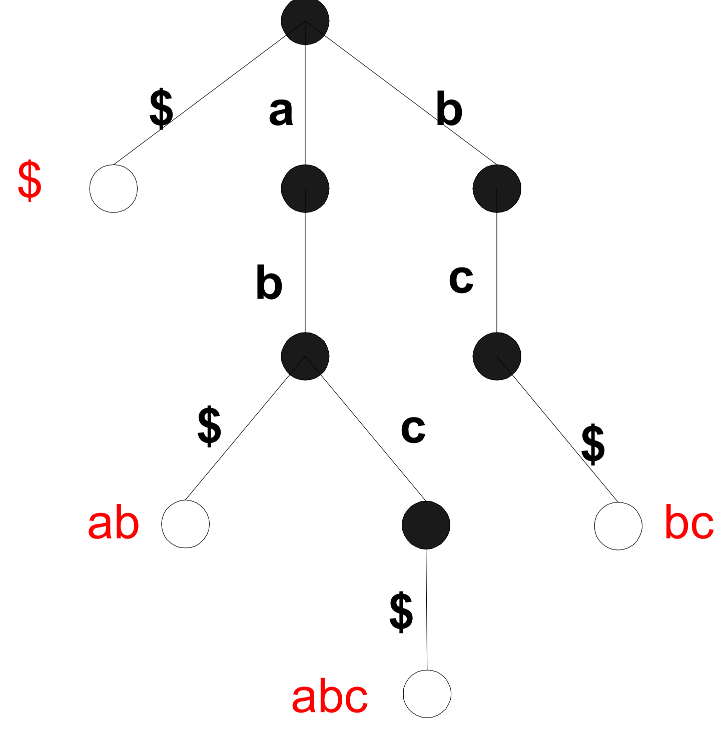
The basic idea -- any substring is a prefix of a suffix:



A common data structure ***trie*** (pronounced *try*) for re***trie***val

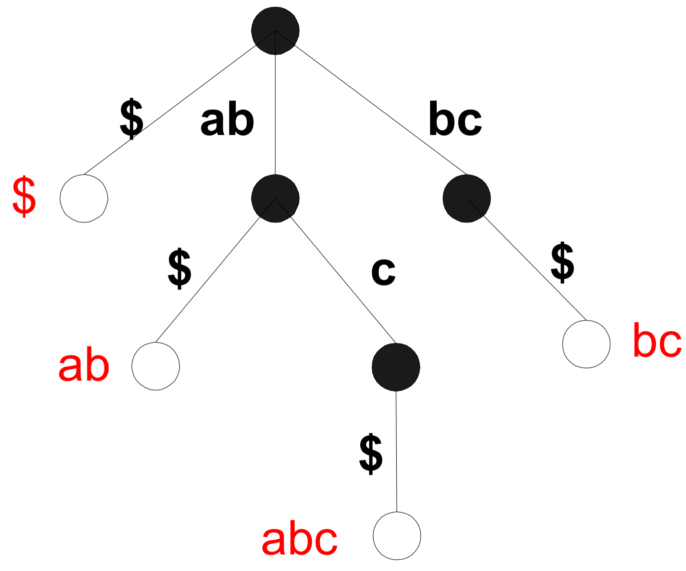
A ***trie*** on a set $X=\{x_1, x_2, .. x_n\}$ of pairwise distinct strings is a search tree with $n+1$ leaves. The edges are labelled by characters, a path to a leaf "spells" a string from X. For technical reasons, each strings is terminated by the lexicographically smallest sentinel symbol $.

(*For* C/C++ *aficionados, think of* $ *as* NULL)



A trie on X = { ab, abc, ba }

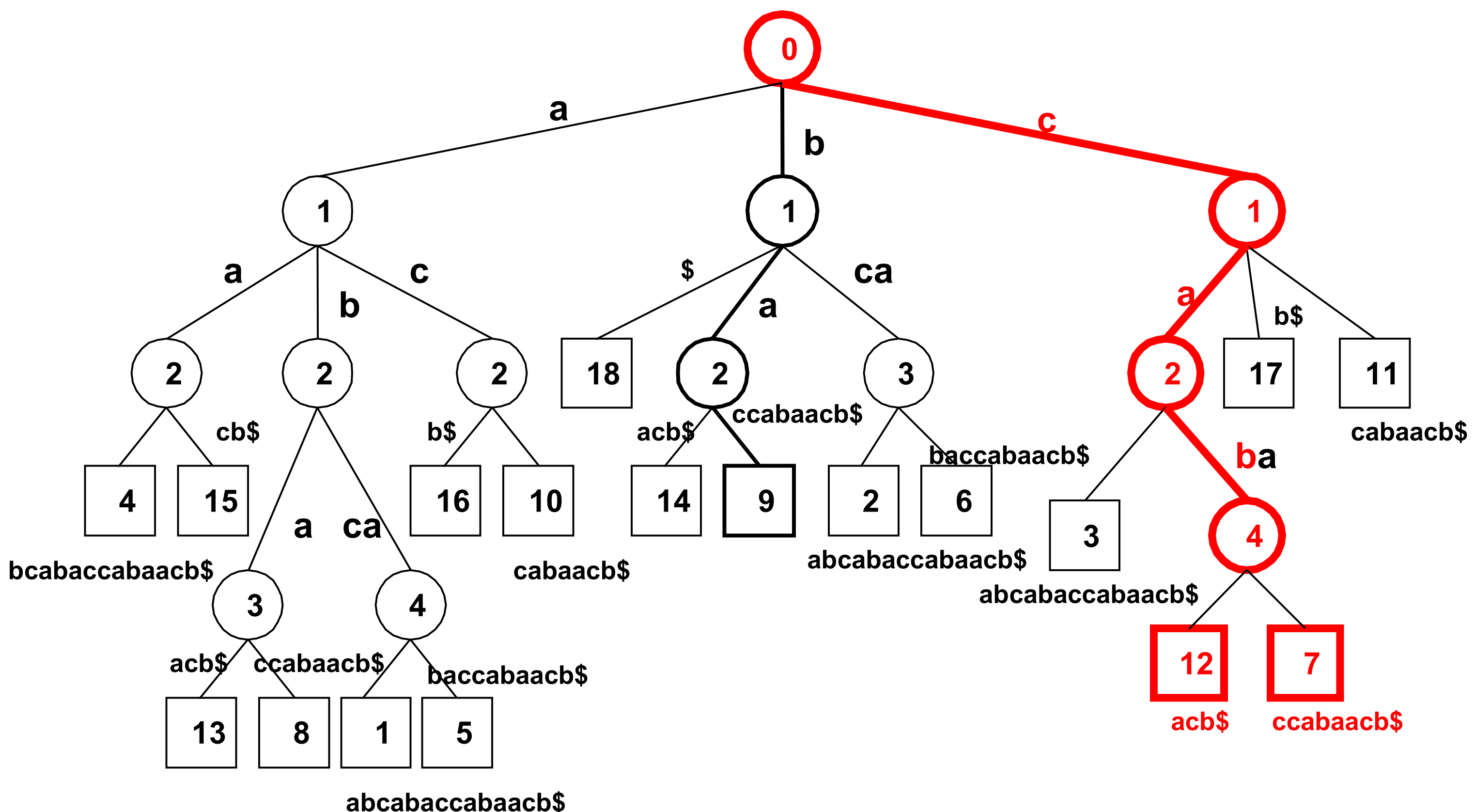
Patricia trie (***compacted trie, radix tree***) all internal nodes of degree 2 are eliminated and the edges are labelled by substrings



A Patricia trie on $X = \{ \text{ab, abc, ba} \}$

(***P****ractical* ***A****lgorithm* ***T****o* ***R****etrieve* ***I****nformation* ***C****oded* ***I****n* ***A****lphanumeric*) *Morrison* (*1968*)

Suffix tree of a string x = Patricia trie of the set of all nontrivial suffixes of x *Weiner* (*1973*)



Applications -- search for substrings in a string:

- Check if a string P of length m is a substring in $O(m \log \alpha)$, where α is the size of the alphabet.
- Find the first occurrence of the patterns $P_1,\ldots,P_q$ of total length m as substrings in $O(m \log \alpha)$.
- Find all k occurrences of the patterns $P_1,\ldots,P_q$ of total length m as substrings in $O((m + k) \log \alpha)$.
- Find the *longest common prefix* between two suffixes in $\Theta(\log \alpha)$ (requires preprocessing of the tree)
- Find all k tandem repeats in $O((n \log n + k) \log \alpha)$

Applications -- determine properties of a string:

- Find the longest common substrings of strings s_1 and s_2 in $\Theta(|s_1|+|s_2|)$.
- Find all k maximal repeats in $\Theta(n+k)$.
- Find the Lempel-Ziv decomposition in $\Theta(\mathrm{n})$.
- Find the longest repeated substrings in $\Theta(\mathrm{n})$.
- Find the most frequently occurring substrings of a minimum length in $\Theta(\mathrm{n})$.

To construct a suffix tree

- naïve iterative algorithm $O(n^2)$

- smarter constructions in $O(n \log n)$ - iterative: *Weiner* (*1973*), *McCreight* (*1976*) -faster and less memory, *Crochemore* (*1981*) - via all maximal repetitions, *Ukkonen* (*1995*) - suffix links, online These are linear, if alphabet size is fixed, i.e. for small alphabets.

- complex construction in $O(n)$ for any indexed alphabet - recursive: *Farach* (*1997*)

The basic ideas of Farach's construction

1 Construct suffix tree for odd suffixes of the input string ***x*** by recursion:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18

a b | c a | a b | c a | b a | c c | a b | a a | c b $

2 4 2 4 3 6 2 1 5

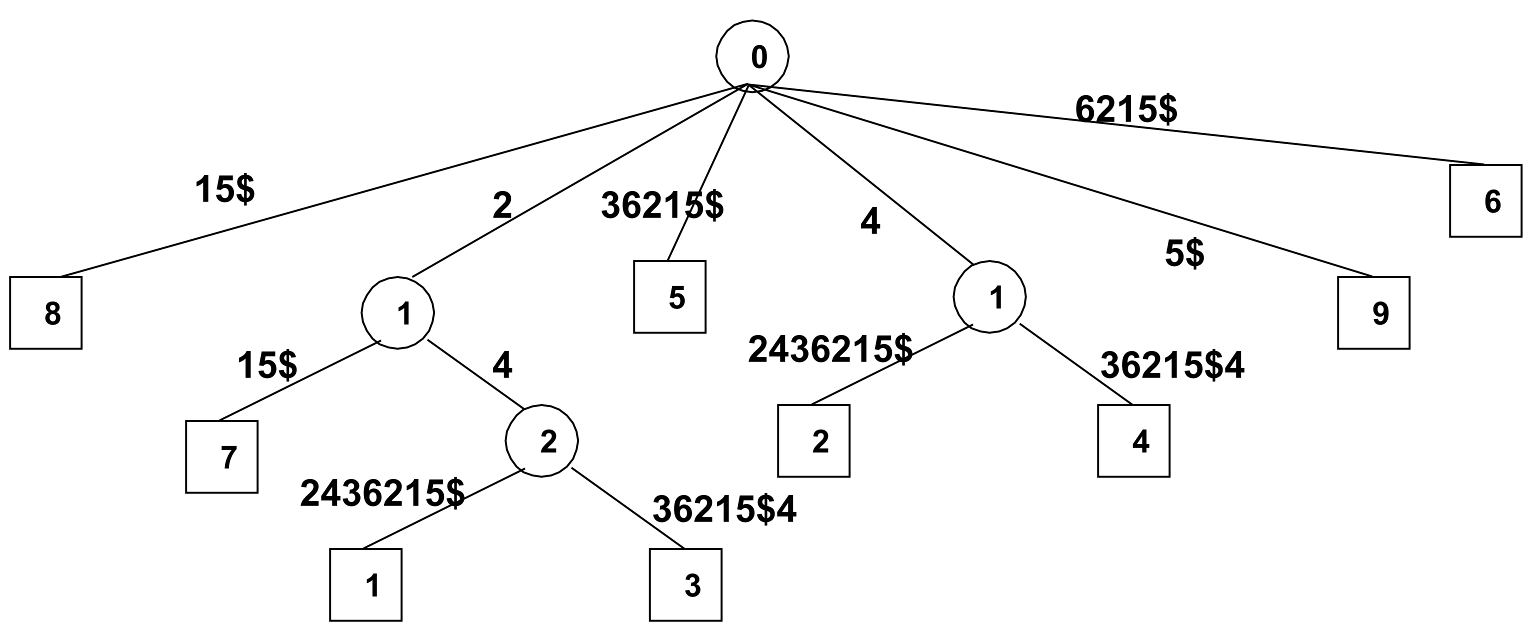
Use radix sort to sort the pairs (x[i],x[i+1]) for odd i:

aa=>1, ab=>2, ba=>3, ca=>4, cb=>5, cc=>6

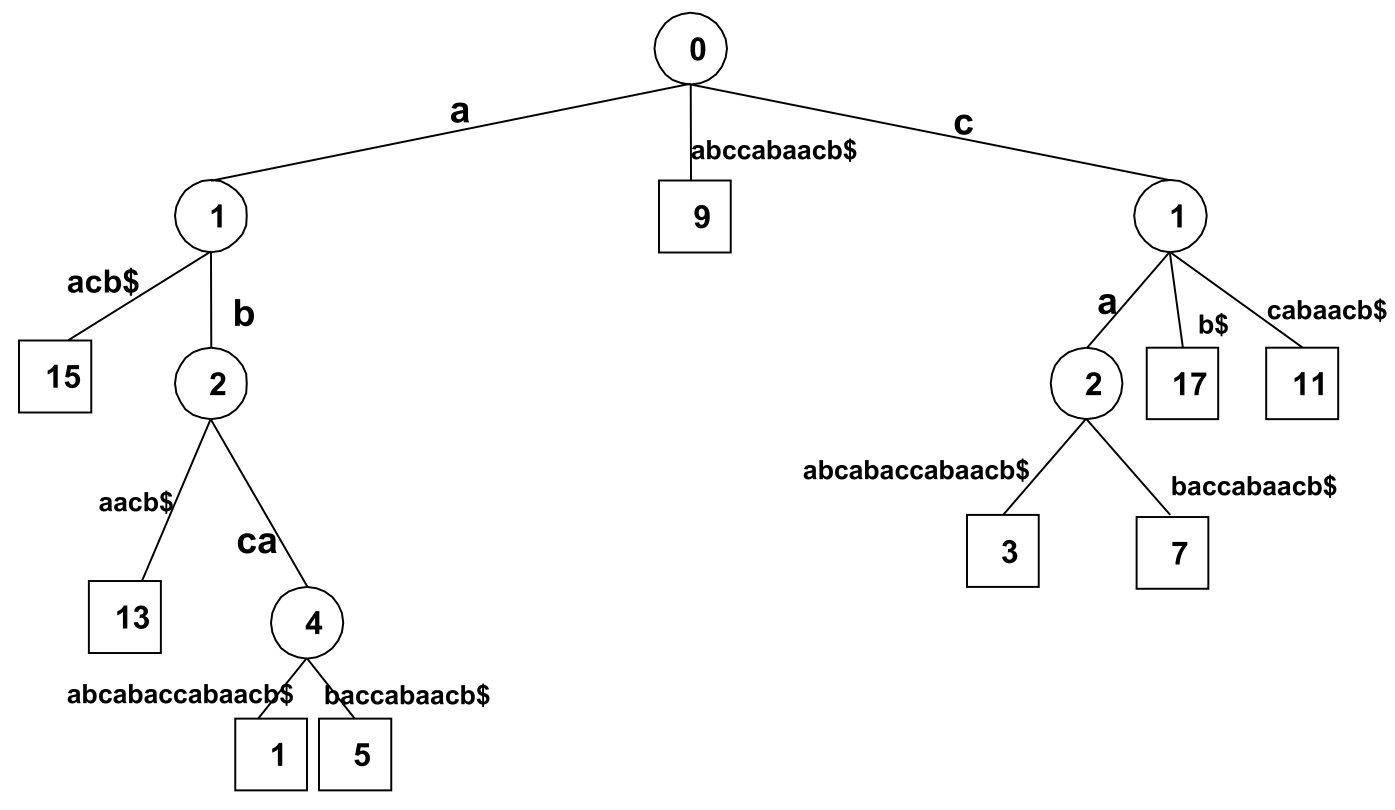
Create a new string ***y*** = 2 4 2 4 3 6 2 1 5 $
and by a recursive call, obtain its suffix tree

1 2 3 4 5 6 7 8 9

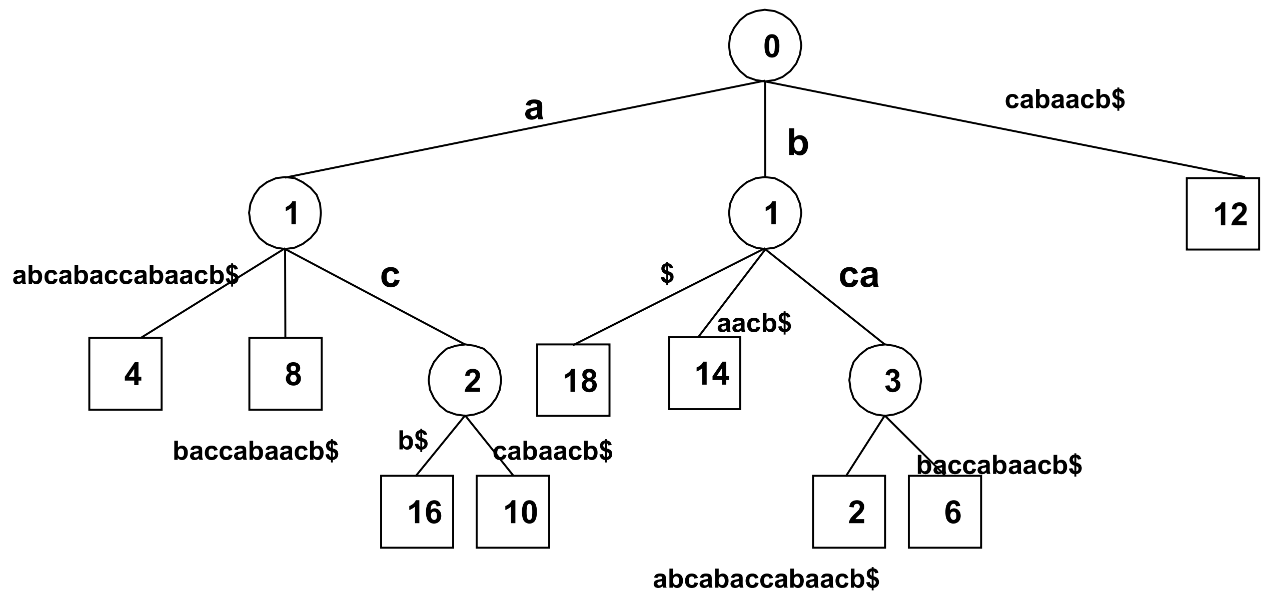
Suffix tree for *y* = 2 4 2 4 3 6 2 1 5 $



Massage it into a suffix tree of odd suffixes of *x* in linear time:



2 From this tree create the suffix tree for even suffixes, also in linear time (again using radix sort):



3 Now merge these two suffix trees into one, also in linear time.

The problem with suffix tree --- too much memory!

5$|\boldsymbol{x}|$ to 10$|\boldsymbol{x}|$ machine words required - *Kurtz* (*1999*) reduced suffix tree!

The construction also requires a lot of additional (working) memory.

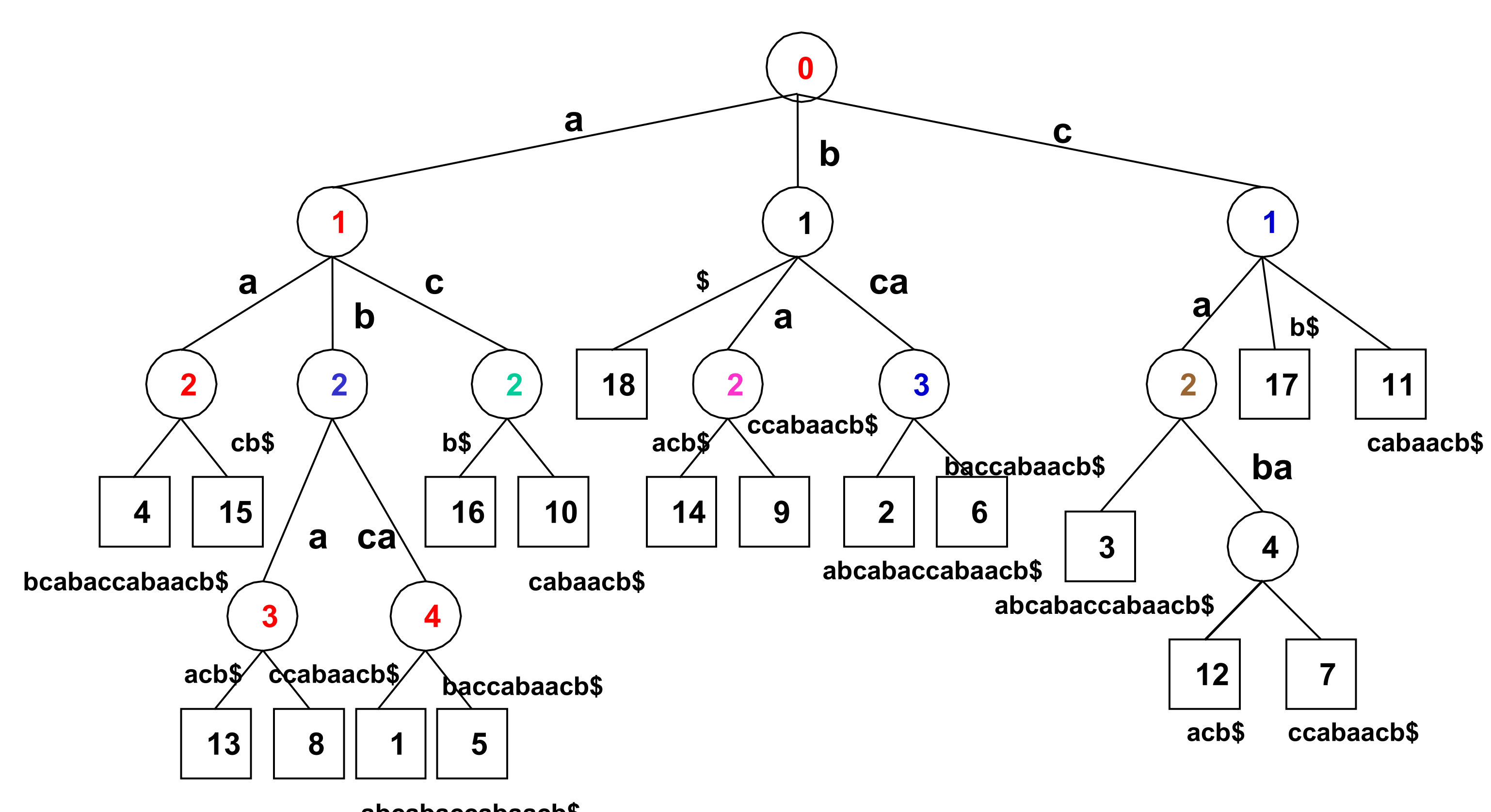
This is unfeasible and impractical for large strings (e.g. DNA - tens/hundreds of millions of "letters").

Manber+Mayers (*1993*) introduced suffix arrays as an alternative to suffix trees.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18

a b c a a b c a b a c c a b a a c b $

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
suff	4	15	13	8	1	5	16	10	18	14	9	2	6	3	12	7	17	11
lcp		2	1	3	2	4	1	2	0	1	2	1	3	0	2	4	1	1



M+M: Search for a substring u in $O(|u| \log n)$, construct ***suff*** in $O(n \log n)$, expected $O(n)$, construct ***lcp*** in $O(n \log n)$, expected $O(n)$.

Kasai et al (*2001*): linear time algorithm to compute ***lcp*** from ***suff***. Problem reduced to suffix sorting.

Abouelhoda et al (*2002*): search for u in $O(|u|)$, with additional linear time preprocessing.

Problems requiring top-down or bottom-up traversal of suffix tree can be performed with the same asymptotic complexity using suffix arrays.

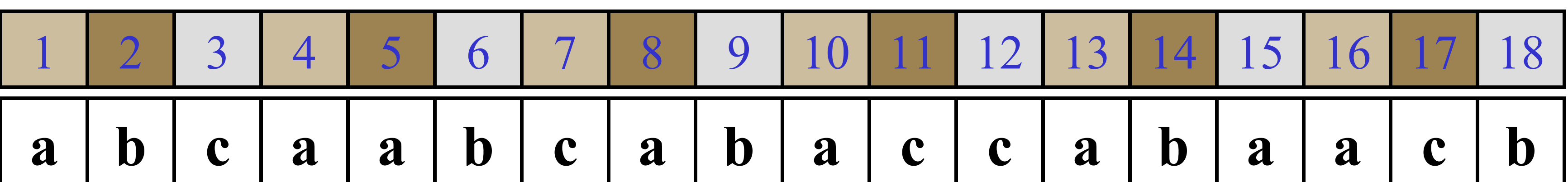
Suffix sorting in linear time

Three papers came out in 2003 giving linear time recursive algorithms for suffix sorting. They all tried "Farach's" approach:

split suffixes into G_1 and G_2

1. *sort G_1 using recursive reduction of the problem*
2. *sort G_2 using the order of G_1*
3. *merge G_1 and G_2*

Kärkkäinen+Sanders: the simplest, the most elegant, the most memory efficient. The question is: how fast?



Suppose to have all brown suffixes ([light brown] and [dark brown]) sorted.

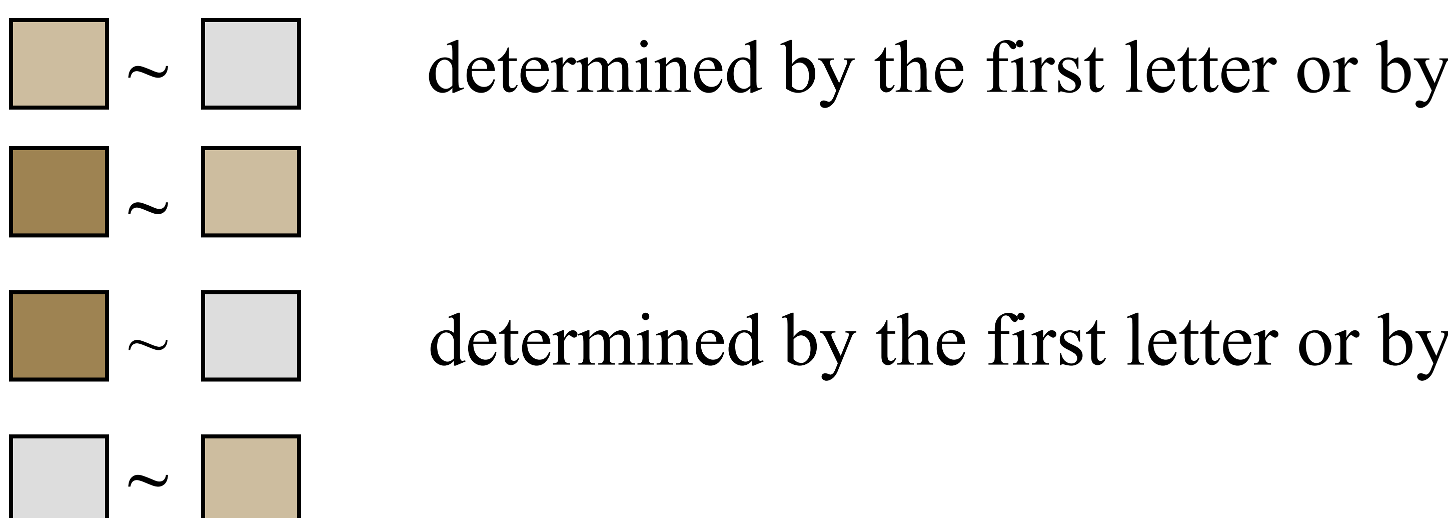
Then [6] ~ [9] determined by $\boldsymbol{x}[6] \sim \boldsymbol{x}[9]$, or if $\boldsymbol{x}[6]=\boldsymbol{x}[9]$, determined by [7] ~ [10]

Thus radix sort with keys of size 2 will do. So we have all gray suffixes sorted.

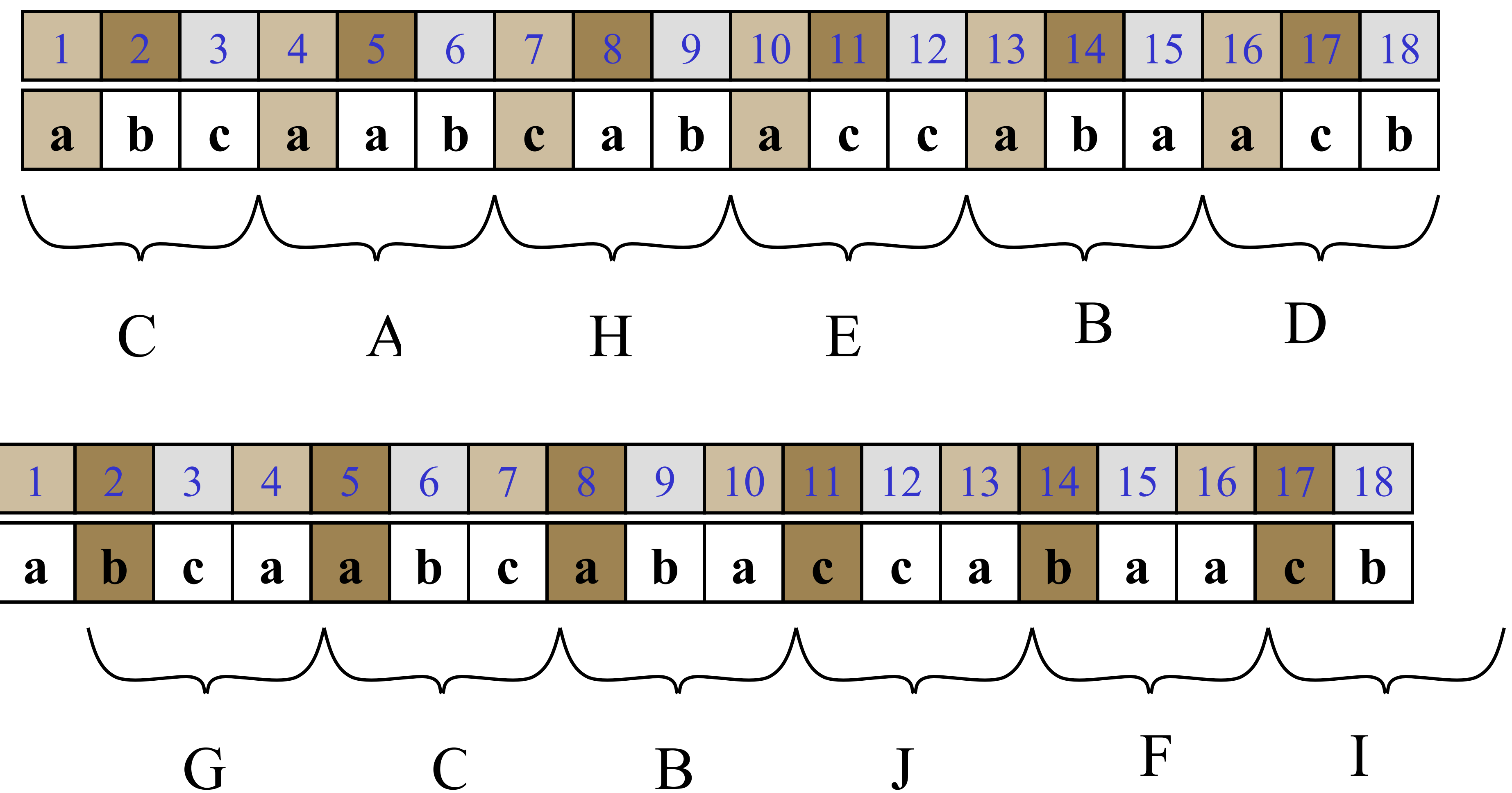
How to merge brown and gray suffixes?

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
a	**b**	**c**	**a**	**a**	**b**	**c**	**a**	**b**	**a**	**c**	**c**	**a**	**b**	**a**	**a**	**c**	**b**

Simple comparison-based merge:



So, how to sort the brown suffixes? Like Farach!



Using radix sort, sort the triples.

1 4 7 10 13 16 2 5 8 11 14 17

C A H E B D G C B J F I

A H E B D G C B J F I 4

B D G C B J F I 13

B J F I 8

C A H E B D G C B J F I 1

C B J F I 5

D G C B J F I 16

E B D G C B J F I 10

F I 14

G C B J F I 2

H E B D G C B J F I 7

I 17

This approach works for any “division” as long as the beige blocks are bigger than the gray blocks. Of course, using bigger beige blocks requires longer radix sort, however it decreases the recursion and memory use.

In many ways, using blocks of size 3 optimizes the solution, see results of a crude simulation:

```
N=100
2+1: total=1415.000000,rec=8,mem=943.333333
3+2: total=1541.866667,rec=6,mem=566.400000
N=1000
2+1: total=14890.000000,rec=14,mem=9926.666667
3+2: total=16202.666667,rec=10,mem=5952.000000
N=10000
2+1: total=149855.000000,rec=20,mem=99903.333333
3+2: total=163209.200000,rec=15,mem=59954.400000
N=100000
2+1: total=1499840.000000,rec=25,mem=999893.333333
3+2: total=1633170.000000,rec=19,mem=599940.000000
N=1000000
2+1: total=14999770.000000,rec=31,mem=9999846.666667
3+2: total=16333150.400000,rec=24,mem=5999932.800000
N=10000000
2+1: total=14999770.000000,rec=31,mem=9999846.666667
3+2: total=16333150.400000,rec=24,mem=5999932.800000
```

Are the linear suffix sorting algorithms practical? They seem to require at least $4|\boldsymbol{x}|$ working memory.
Larsson+Sadakane (*1999*): sorting suffixes as independent strings - for most real-world data very fast, though worst-case complexity is $\Omega(n^2)$, requires very little extra space (for instance **bzip2** by *Seward*).

Manzini+Ferragina (*2002*): very fast, very little extra memory ($0.03n$), however worst-case complexity is $\Omega(n^2)$.

They posed a problem:
lightweight ($O(n \log n)$, sublinear memory) algorithm?

Burkhardt+Kärkkäinen (*2003*): an $O(n \log n)$ suffix sorting algorithms with $O(n / \sqrt{\log n})$ memory requirement. Based on the idea of **difference covers** (VLSI design, distributed mutual exclusion -- *Colbourn+Ling* (*2000*)).

For any pair of suffixes $\boldsymbol{x}[i..n]$, $\boldsymbol{x}[j..n]$ find the smallest k such that the order of $\boldsymbol{x}[i+k..n]$ and $\boldsymbol{x}[j+k..n]$ is known (*anchor pair*).

A *difference cover D modulo v*: set of integers $0..v$-1 such that for any $0 < i < v$ there are $i_1, i_2 \in D$ so that $i = i_1 - i_2 \pmod v$. For $\forall\, i, j$ compute $k = \delta(i, j) \in [0,v)$ so that $((i+k) \bmod v)$, and $((j+k) \bmod v)$ are both in D (can be done in $O(v)$).

Then sort all suffixes whose starting position is in D. The sort of all suffixes is transformed to a sort on keys of length $\leq v+1$.

Note that *Kärkkäinen+Sanders* algorithm uses D *modulo* 3!

Colbourn+Ling (*2000*): For every v, a difference cover D *modulo* v of size $|D| \leq \sqrt{1.5v}+6$ can be computed in $O(\sqrt{v})$ time.

Currently the fastest linear suffix sorting algorithm is due to *Maniscalco+Puglisi* 2008.

Can suffix array really "replace" the string?
Bannai, Inenaga, Shinohara, and Takeda (*2003*): given an array, conditions can be checked if it is a suffix array of a string (must be a permutation of n) and such a string with a minimal alphabet is inferred from the array in $O(n)$ time.